Energy- and Quantization-aware DNN Partitioning in the Edge-Cloud Continuum (Work In Progress Paper)

Simone Nicosanti
nicosanti@ing.uniroma2.it
Tor Vergata University of Rome
Rome, Italy

Gabriele Russo Russo
russo.russo@ing.uniroma2.it
Tor Vergata University of Rome
Rome, Italy

Valeria Cardellini
cardellini@ing.uniroma2.it
Tor Vergata University of Rome
Rome, Italy

Abstract

Deep neural networks (DNNs) are pervasive across various domains, with inference requests often generated at the network edge, where resources are limited and energy efficiency is critical. Techniques like Post-Training Quantization (PTQ) also emerged to facilitate inference at the edge, trading off resource demand with accuracy. However, running inference entirely on devices can lead to high latency and excessive battery drain, while executing it exclusively in the cloud introduces communication delays and may result in a significant environmental impact. As such, inference tasks must carefully exploit both edge and cloud computing resources, leveraging DNN model splitting (or partitioning).

In this work, we present a multi-objective optimization problem to distribute DNN model inference across the edge–cloud continuum while integrating PTQ. We develop a prototype architecture to profile DNN models and the underlying computing infrastructure, and we address the issue of estimating quantization noise. Evaluated on YOLO11 vision models, our approach achieves significant reductions in both inference times and energy consumption (up to 30% for both metrics) compared to device-only inference execution.

CCS Concepts

• **Computing methodologies** → **Distributed artificial intelligence**; **Distributed computing methodologies**; • **Hardware** → **Power and energy**.

Keywords

Deep learning, edge computing, model quantization

ACM Reference Format:
Simone Nicosanti, Gabriele Russo Russo, and Valeria Cardellini. 2026. Energy- and Quantization-aware DNN Partitioning in the Edge-Cloud Continuum (Work In Progress Paper). In *Companion of the 17th ACM/SPEC International Conference on Performance Engineering (ICPE Companion '26), May 04–08, 2026, Florence, Italy*. ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3777911.3801106

1 Introduction

The integration of Deep Learning (DL) models within Artificial Intelligence (AI) pipelines is becoming ubiquitous across various domains (e.g., speech recognition, autonomous driving, image analysis [3]). While model training is usually executed in the cloud using powerful GPU clusters, inference requests are frequently generated at the network edge, by devices like sensors, smartphones or drones, where resources are limited and energy efficiency is critical. These devices may struggle to sustain inference workloads alone due to limited computing capacity and (possibly) excessive battery drain, especially as AI models rapidly grow in size and complexity. At the same time, serving inference requests only in the cloud is suboptimal due to data transfer overheads, privacy concerns, and the environmental impact of data center carbon footprints [13].

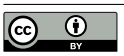
A viable approach to address these challenges is to *split* (or *partition*) DNN models and distribute their layers across a spectrum of resources, from edge devices to cloud servers, to balance latency, energy consumption, and performance. This approach may be used – for instance – to move more resource-demanding layers to cloud servers, while keeping the rest of the DNN on local device. However, optimal model splitting requires careful consideration of network conditions and heterogeneous capabilities of different hardware, including their energy requirements.

In addition to model splitting, several techniques have been proposed to accelerate inference workloads at the edge [16]. Among them, model approximation aims to reduce inference costs accepting (limited) accuracy degradation. In particular, Post-Training Quantization (PTQ), which reduces the numerical precision of network weights and/or activations, is appealing as it accelerates computation and reduces data size without model re-training [6]. Nevertheless, PTQ introduces quantization noise, which may degrade the accuracy of the model and therefore must be carefully managed.

This work aims to optimally distribute DL inference across the edge–cloud continuum while integrating PTQ. The goal is to jointly minimize inference time and energy consumption, subject to constraints on maximum quantization noise and edge device energy budget. The proposed approach takes into account device heterogeneity, as well as computational and transmission costs, and, differently from related works [10, 11], considers the effects of quantization on both execution time and data transfer.

Our key contributions can be summarized as follows:

- We present a holistic framework for the deployment of DL models in the edge–cloud continuum, which, starting from model and network characteristic profiles, automatically optimizes distributed inference tasks (Sec. 3).
- We propose an efficient data-driven approach to model the quantization noise (Sec. 4).
- We formulate the optimal deployment problem as an Integer Linear Programming (ILP) problem, jointly considering model splitting and quantization (Sec. 5).
- By deploying our framework on a small testbed in a public cloud infrastructure, we demonstrate the effectiveness of our



ICPE Companion '26, Florence, Italy

ACM ISBN 979-8-4007-2326-1/2026/05
https://doi.org/10.1145/3777911.3801106

approach and the impact of jointly modeling splitting and quantization, considering YOLO11 models [7] (Sec. 6).

As a work in progress, this paper highlights a promising direction that can be further extended, as discussed in Sec. 7.

2 Related Works

The computational challenges of performing DL inference at the network edge have motivated research works for more efficient workload distribution [12], as well as techniques to trade off accuracy for reduced computational demand, such as model pruning and quantization [9], early exit, or compression [15], as surveyed in [17]. We briefly review the approaches closest to ours that leverage model splitting and (possibly) quantization.

Model splitting (or partitioning) has emerged as a key technique to distribute DL inference workloads across multiple infrastructure layers [12, 17]. Most research focuses on a two-layer architecture, typically involving only edge devices and cloud servers, and address model splitting using various methodologies, including optimization theory, heuristics, and reinforcement learning. Earlier works dealt with sequential DNNs (e.g., [8]), while more recent studies address complex directed acyclic graphs (e.g. [2]). The works most relevant to our study are [4, 5, 8]. In [8] predictive models estimate, for a given device and layer type, the time and energy required to execute that layer on the device. The partition point is evaluated after each layer, splitting the DNN into a "head" executed locally and a "tail" executed in the cloud, on the basis of latency or energy consumption, factoring in transmission time. In [4] a joint device-cloud inference framework minimizes energy consumption while ensuring that inference time remains below a threshold. The model is split into consecutive layers, and the optimal allocation is found through a shortest-path problem on a decision graph. In [5] a more complex scenario is considered with multiple paths between a device and edge nodes. Intermediate nodes can execute parts of the model, and the goal is to find the optimal path and layer allocation to minimize inference time. The problem is modeled as an adversarial multi-armed bandit problem, allowing for online learning of the optimal path even under uncertain network conditions. In contrast to these works, we consider an edge-cloud continuum scenario, where the DNN layers can be split across multiple computing layers, rather than just between two layers, minimizing inference time and energy consumption.

Works that leverage both model splitting and quantization have also been proposed [2, 10, 11]. In [10], the optimization phase focuses on finding the best partition point of the DNN to minimize the total inference time; once the network is split into the edge and the cloud parts, the edge part is quantized. Auto-Split [2] relies on an optimization problem to identify the split between edge device and cloud server and the bit-width assignment for weights and activations, but it does not consider energy aspects, as we do. In [11] a formulation similar to [18] is used to find the optimal precision for both weights and activations of (mainly) linear classifiers, while also proving the linearity of the noise induced by quantization.

Compared to [10], our approach considers the impact of quantization in terms of both computation and transmission time during the optimization phase. Furthermore, unlike [11], our approach can handle complex DNN topologies, such as YOLO11 models.

3 Proposed Architecture

We propose a framework to optimally serve DNN inference requests in the edge-cloud continuum. Starting from ONNX model format and ONNX Runtime, our framework introduces essential components to profile, optimize and deploy inference workloads in a distributed manner. Figure 1 illustrates a high-level representation of the framework, with the six core components described below.

Model Profiling. We extract the main characteristics of the input DNN, modeling it as a graph; for example, we extract the names of the layers or information about the tensors. We also build a polynomial regression to predict the quantization noise, as better explained in Sec. 4.

Network Profiling. We model the server network's characteristics as a graph, for example, extracting latency and bandwidth information. We also collect information about the servers in the system, such as their energy characteristics.

Execution Profiling. For each layer of the DNN, we profile the time taken to run it on each server; as we are considering quantization, the profiling is done for both the standard and quantized versions of the layer. Moreover, in order to reduce the overhead given by executing separated layers, we profile the time taken to run the whole model, both in its not quantized and mixed versions.

Plan Generation. We gather the profiles generated in previous phases in order to generate a deployment plan. This phase can be divided into two phases. In the *Optimization Phase*, we use the profiles to build and solve an optimization problem. The output of this first phase is a group of layers assignments, as well as the decision regarding which layers should be quantized. In the *Post-processing Phase*, starting from the assignments and from the quantization decisions, we build a graph of components, where each component represents a sub-model of the original one. We then generate a deployment plan containing the main information about the structure of the components graph, as well as the information regarding the layers that should be quantized.

Plan Actuation. According to what is indicated in the deployment plan, we first generate a version of the model with quantized layers, and then we split the model, breaking it into sub-models. Finally, we provide each server in the system with the plan and the components assigned to it.

Inference. Following the optimized plan, the servers in the system can collaborate in the inference process, exchanging the intermediate results of their sub-models until obtaining the final output of the model.

4 Quantization Noise Modeling

In our modeling, we define the quantization noise as follows:

$$\rho = \frac{1}{A} \sum_{i=1}^{A} \frac{1}{m} \parallel \mathbf{o}_i - \mathbf{o}_{i,q} \parallel_1 \tag{1}$$

where A is the size of the dataset, m is the dimensionality of the output, $\mathbf{o}_i$ and $\mathbf{o}_{i,q}$ are the outputs of the original and quantized models on i-th sample.

Modeling noise becomes increasingly complex as the model size grows: in fact, for a DNN with L layers, considering even a single variant of quantization results in 2^L possible versions of the quantized model. Therefore, to limit the resulting complexity, we

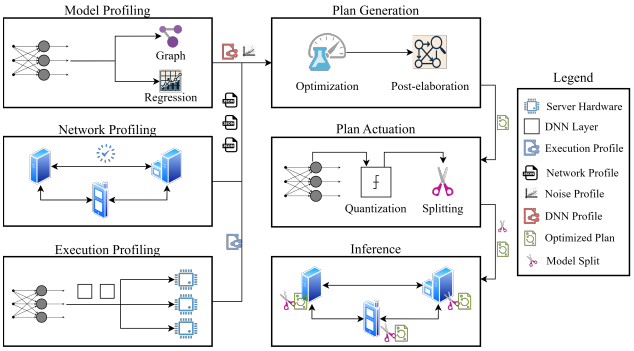


Figure 1: Overview of our framework.

focus on the subset of layers that can benefit most from quantization, i.e., the layers with the maximum computational demand, measured in FLOPS.

Assuming there are L_Q quantizable layers and denoting V_Q as the subset of these layers, we model the quantization noise as follows:

$$\rho(\mathbf{q}) = \sum_{C \in \mathcal{P}(V_Q)} \left(\rho_C \cdot \prod_{i \in C} q_i \cdot \prod_{i \notin C} (1 - q_i) \right) \tag{2}$$

where ρ_C is the noise for the quantization configuration C, q_i represents whether the i-th layer is quantized (with $q_i \in \{0, 1\}$), and $\mathcal{P}(V_Q)$ is the power set of V_Q.

However, to integrate such formulation into an optimization problem, we would need to compute ρ_C for all the possible configurations. To avoid doing so, we approximate the value of $\rho(\mathbf{q})$ training a polynomial regressor. Specifically, we randomly build a subset of the 2^{L_Q} quantization combinations and evaluate the quantization noise for each of the resulting quantized models. The produced training set is used to train the regressor $\eta(\mathbf{q})$:

$$\eta(\mathbf{q}) = \sum_{C \in \mathcal{P}_{\leq d}(V_Q)} \left(\lambda_C \cdot \prod_{i \in C} q_i \right) \tag{3}$$

where d is the degree of the polynomial, $\mathcal{P}_{\leq d}(V_Q)$ is the power set of V_Q restricted to the elements of cardinality less or equal to d, and λ_C is the coefficient for a combination C in $\mathcal{P}_{\leq d}(V_Q)$.

Such a predictor can be integrated into an optimization problem much more easily than the exact model, as we can control the complexity of the polynomial with its degree d, and the product of variables is much less complex. Moreover, as we are training a polynomial, we do not need the noise for all combinations, but just for a subset: clearly, the more data we use for training, the more accurate the predictions will be.

In Table 1 we present an example of regression fitting applied to the raw output of YOLO11n-det model with *INT8* quantization. In this experiment, we considered 10 quantizable layers, a calibration set of 100 samples, a noise evaluation set of 10 samples, a train set of 500 samples, and a test set of 50 samples. The result shows that the performance of the polynomial fitting improves as the degree increases, reaching its maximum at the 4th degree, and showing the first signs of overfitting starting from the 5th degree.

Table 1: R^2-scores as a function of polynomial degree

	Deg 1	Deg 2	Deg 3	Deg 4	Deg 5
Train Score	0.8711	0.9759	0.9989	1.0	1.0
Test Score	0.8830	0.9784	0.9984	0.9998	0.9950

Since the third-degree polynomial achieved good scores on both the training and test sets, we selected this degree for the experiments in Sec. 6, as it provides a good tradeoff between noise prediction accuracy and polynomial complexity.

The strategy we presented aims to model quantization noise on the DNN output rather than the drop in accuracy. While end users are most concerned with the latter, the former is more general, as it can be easily adapted to a wide range of tasks and does not depend on the availability of labeled validation data. In general, we expect the quantization noise to be sufficiently correlated with accuracy degradation to serve as a reliable indicator [1, 14]. In what follows, we consider only INT8 quantization, as it is broadly supported by various hardware platforms and by the inference framework we have chosen, namely ONNX Runtime.

5 Optimization Problem

In this section, we present the optimization problem we formulated to optimally partition the DNN across computing layers, taking into account inference time, energy consumption, and the potential benefit of quantization.

5.1 Base Model

We model the DNN and the server network as two separate graphs, as follows:

- The DNN is a directed acyclic graph (DAG) $G_D = (V_D, E_D)$ with additional information T_D regarding tensors transmitted among its layers. To add, let $V_I \subset V_D$ and $V_O \subset V_D$ be the input and output layers of the DNN, respectively.
- The server network is a mesh $G_N = (V_N, E_N)$ with loops (i.e., $\forall k \in V_N, (k, k) \in E_N$).

We model the problem as a graph assignment problem, considering both computation and energy aspects. For each, we account for the contributions from both computation and transmission, as well as the benefits provided by quantization.

The main decision variables of the optimization problem are:

- $x_{ik} \in \{0, 1\} \quad \forall i \in V_D, \forall k \in V_N$; $x_{ik} = 1$ iff layer $i \in V_D$ is assigned to server $k \in V_N$;
- $y_{tn} \in \{0, 1\} \quad \forall t \in T_D, \forall n \in E_N$; $y_{tn} = 1$ iff tensor $t \in T_D$ is transmitted through network link $n \in E_N$;
- $q_i \in \{0, 1\} \quad \forall i \in V_D$; $q_i = 1$ iff layer $i \in V_D$ is quantized;
- $x^q_{ik} \in \{0, 1\} \quad \forall i \in V_D, \forall k \in V_N$; $x^q_{ik} = 1$ iff layer i is assigned to server k and quantization is activated for layer i;
- $y^q_{tn} \in \{0, 1\} \quad \forall t \in T_D, \forall n \in E_N$; $y^q_{tn} = 1$ iff tensor t is transmitted through network link n and the generator layer of tensor t is quantized.

5.2 Time and Energy Model

5.2.1 Computation Time. We measure the processing time for (possibly quantized) layer i on server k as part of the *Execution Profile*

phase: let this time be $f_k(i, q_i)$. Consequently, we can model the time taken to run a layer as follows:

$$\begin{aligned} T^c_{ik} &= f_k(i,0) \cdot x_{ik} - (f_k(i,0) - f_k(i,1)) \cdot x_{ik} \cdot q_i \\ &= f_k(i,0) \cdot x_{ik} - (f_k(i,0) - f_k(i,1)) \cdot x^q_{ik} \end{aligned} \tag{4}$$

where the term in parentheses can be interpreted as a *quantization gain* and where the x^q_{ik} variable is subject to the linearization constraints of product, which are:

$$x^q_{ik} \le x_{ik}; \qquad x^q_{ik} \le q_i; \qquad x^q_{ik} \ge x_{ik} + q_i - 1 \tag{5}$$

As a result, the computation time of server k T^c_k and the total computation time T^c can be computed as follows:

$$T^c_k = \sum_{i \in V_D} T^c_{ik} \qquad T^c = \sum_{k \in V_N} T^c_k \tag{6}$$

5.2.2 Transmission Time. Considering the following:

- t a tensor of the DNN, $t \in T_D$;
- n a network link, $n \in E_N$, with $n[0]$ and $n[1]$ its source and sink nodes;
- $g(t, n)$ the time taken to transmit the tensor t through the network link n;
- $i \in V_D$ the source layer of tensor t.

We measure the bandwidth and round-trip time (RTT) of network link n as part of the *Network Profile* phase; let $g(t, n)$ be:

$$g(t,n) = \frac{s_t}{b_n} \tag{7}$$

where s_t is the size of tensor t in megabyte (MB) and b_n is the bandwidth of the network link n in MB per second.

Therefore, we can model the time taken to transmit tensor t through link n as follows:

$$\begin{aligned} T^x_{tn} &= (g(t,n) + r_n) \cdot y_{tn} - \left(g(t,n) - \frac{g(t,n)}{\gamma}\right) \cdot y_{tn} \cdot q_i \\ &= (g(t,n) + r_n) \cdot y_{tn} - \left(g(t,n) - \frac{g(t,n)}{\gamma}\right) \cdot y^q_{tn} \end{aligned} \tag{8}$$

where r_n is the RTT of the network link n and γ is the data size scaling factor due to quantization, which is $\gamma = 4$ in our case, as we are considering *FLOAT32* to *INT8* quantization.

Additionally, the term in parentheses can be seen as a *quantization gain* and the y^q_{tn} variable is subject to the linearization constraints of product, which are:

$$y^q_{tn} \le y_{tn}; \qquad y^q_{tn} \le q_i; \qquad y^q_{tn} \ge y_{tn} + q_i - 1 \tag{9}$$

The transmission time T^x_k of a server k is computed as follows:

$$\begin{aligned} T^x_k &= \sum_{t \in T_D} \left(\sum_{n \in E_N \wedge k = n[0] = n[1]} T^x_{tn} \quad + \sum_{n \in E_N \wedge k = n[0] \wedge k \neq n[1]} T^x_{tn} \right) \\ &= \sum_{t \in T_D} \left(T^{x-self}_{tk} + T^{x-other}_{tk} \right) = T^{x-self}_k + T^{x-other}_k \end{aligned} \tag{10}$$

where T^{x-self}_k and $T^{x-other}_k$ are the transmission times to itself and to other nodes, respectively.

Thus, the total transmission time is given by:

$$T^x = \sum_{k \in V_N} T^x_k \tag{11}$$

5.2.3 Time. The total time T is computed as $T = T^c + T^x$.

5.2.4 Energy. We assume that the energy consumption of server k to complete task j (denoted as E^j_k) is linear with respect to the time required to perform that task, with α^j_k as the proportionality coefficient. We thus model the energy consumption as follows:

$$E^j_k = \alpha^j_k \cdot T^j_k \qquad E_k = \sum_{j \in \{c, x-self, x-other\}} E^j_k \qquad E = \sum_{k \in V_N} E_k \tag{12}$$

5.3 Quantization Noise Model

As described in Sec. 4, to avoid an explosion in problem size, we consider only a subset of the model layers as quantizable. Let:

- $V_Q \subset V_D$ be this subset of layers;
- $\mathbf{q} = \{q_i\}_{i \in V_Q}$ be the vector of quantization decision variables of the problem for layers in V_Q;
- $\eta(\mathbf{q})$ be a d-degree polynomial regression modeling the quantization noise, as defined in Eq. 3;
- $\mathcal{P}_{1 \le |\cdot| \le d}(V_Q)$ be the power set of V_Q restricted to the elements of cardinality less or equal to d and greater or equal to 1.

To model the problem linearly, we need to linearize the products inside $\eta(\mathbf{q})$. Let $\hat{q}_k \in \{0, 1\}$ be a variable representing the logical product of the variables q_i for $i \in V_{Q_k}$, where $V_{Q_k} \in \mathcal{P}_{1 \le |\cdot| \le d}(V_Q)$. Each of these variables is subject to the following constraints:

$$\hat{q}_k \le q_i \quad \forall i \in V_{Q_k}; \qquad \hat{q}_k \ge \sum_{i \in V_{Q_k}} q_i - (|V_{Q_k}| - 1) \tag{13}$$

Moreover, we define a variable $p \in \{0, 1\}$ where $p = 1$ iff $\exists i \in V_Q$ such that $q_i = 1$; this variable is subject to the following constraints:

$$p \ge q_i \quad \forall i \in V_Q; \qquad p \le \sum_{i \in V_Q} q_i \tag{14}$$

Let vector $\hat{\mathbf{q}}$ be $\hat{\mathbf{q}} = \{\hat{q}_k\}_{V_{Q_k} \in \mathcal{P}_{1 \le |\cdot| \le d}(V_Q)}$; we can linearize the polynomial as follows:

$$\hat{\eta}(\hat{\mathbf{q}}, p) = \boldsymbol{\lambda}^T \hat{\mathbf{q}} + c \cdot p \tag{15}$$

where $\boldsymbol{\lambda}$ is the vector of polynomial coefficients and c is the constant term. Note that $\hat{\eta}(\mathbf{0}, 0) = 0$: since no layers are quantized, the model is identical to the original one and there is no noise on the output.

5.4 Constraints

5.4.1 Quantization. We assume that only a subset of layers V_Q can be quantized, and thus:

$$q_i = 0 \quad \forall i \notin V_Q \tag{16}$$

5.4.2 Assignment. We enforce that (i) each layer is assigned to exactly one server, and (ii) input and output layers are assigned to the server initiating the inference process:

$$\sum_{k \in V_N} x_{ik} = 1 \quad \forall i \in V_D; \qquad x_{i0} = 1 \quad \forall i \in V_I \cup V_O \tag{17}$$

5.4.3 Flow. We must ensure that a tensor is transmitted over a network link if and only if both of the following conditions hold: the tensor source layer is placed on the source node of the link *and* at least one of the tensor destination layers is placed on the destination node of the link. We get:

$$y_{tn} \leq x_{ik}; \quad y_{tn} \leq \sum_{j \in V_D^t} x_{jh}; \quad y_{tn} \geq x_{ik} + \frac{1}{|V_D^t|} \sum_{j \in V_D^t} x_{jh} - 1 \quad (18)$$
$$\forall t \in T_D, \forall n = (k,h) \in E_N$$

where V_D^t is the set of layers receiving tensor t as input.

5.5 Problem Formulation

Given the model and constraints introduced above, the resulting optimization problem is defined as follows:

$$\begin{aligned} min \quad & w_T \cdot T^{norm} + w_E \cdot E^{norm} \\ subject \ \ to \quad & E_0 \leq J_0 \\ & \hat{\eta}(\hat{\mathbf{q}}, p) \leq \eta_{max} \end{aligned} \quad (19)$$

together with the constraints defined in Sec. 5.4. We further define:

- T^{norm} and E^{norm} as the min–max normalized versions of T and E, respectively;
- w_T and w_E as the weights associated with inference time and energy consumption, with $w_T + w_E = 1$;
- J_0 as the energy limit of server $k = 0 \in V_N$, which we assume initiates the inference process;
- η_{max} as the upper bound on the quantization noise.

5.6 Solution Post-processing

Once the optimal solution is available, we build subgraphs (or, components) of the original DNN graph based on layer assignments to servers. In this phase, dependency cycles may arise between subgraphs, as there is no acyclicity constraint enforced in the problem formulation. We break such cycles by tracking dependencies and possibly splitting components (see Appendix A for more details).

Once the components are defined, we build the deployment plan. In this plan, for each component, we indicate the server it is assigned to and we define the input and output tensors names; this last information is pivotal in the actuation phase in order to split the model. We also list the layers that must be quantized, according to the values of the decision variables.

6 Experiments

6.1 Experimental Setup

We considered three computing layers (device, edge, and cloud) among which the DNN layers can be split, and deployed our framework on Google Cloud Platform (GCP) using Docker to account for heterogeneous hardware capabilities and limited resource availability (with one CPU allocated for both device and edge). For the network setup, we used `tc` to emulate bandwidth and latency. Tables 2 and 3 report the hardware and network settings.

In our evaluation, we focused on YOLO11 family models [7], specifically the semantic segmentation model YOLO11x-seg (62.1 M parameters, 320.2 GFLOPs), which is one of the most complex models of the family. This model is made up of 652 layers, with two additional layers for input and output. We imported the models in ONNX format and used ONNX Runtime as execution framework. For quantization, we used the APIs provided by ONNX Runtime, considering a maximum of 12 quantizable layers, which we found to be a good compromise between the number of possible combinations (and the resulting problem complexity) and the benefit achievable through quantization. To solve the optimization problem, we used IBM CPLEX.

Table 2: Hardware setup

	Machine	GPU	Comp. power	Tx. power
Device	e2-standard	-	2.9165 W	3.507 W
Edge	c3-standard	-	5.833 W	2.265 W
Cloud	n1-standard	Tesla T4	35 W	0.014 W

Table 3: Network setup (bandwidth and latency)

	Max BW	Lat to Device	Lat to Edge	Lat to Cloud
Device	5 MB/s	-	5 ms	55 ms
Edge	20 MB/s	5 ms	-	50 ms
Cloud	100 MB/s	55 ms	50 ms	-

6.2 Results

We now present the results obtained by running our framework. In order to obtain them, we first executed the profiling phases on the target DNN and on the servers; then we solved the problem for different optimization parameters, thus obtaining the optimized plan and applying it. As mentioned earlier, we used the total model runtime on each device to normalize the per-layer runtimes and quantization gain for that device.

The results are averaged over 25 runs, measured by deploying the framework in GCP. Energy consumption is estimated using the linear model presented in Sec. 5.2, since GCP does not provide tools for directly measuring it.

Our aim is to validate the benefit provided by our framework; therefore, the baseline we consider is executing the model on just the device with no quantization applied.

6.2.1 Latency Evaluation. In Fig. 2 we show the results obtained when optimizing latency (since the standard deviation was negligible, error bars are not shown). For both the *device* and *device+edge* configurations, we can observe how the quantization improves the inference time. In the first case, the time decreases from 5.27 s to 3.27 s (a 30.3% improvement), while in the latter it decreases from 3.64 s to 2.42 s (a 33.5% improvement). These improvements are surprising if we consider that in the last noise configuration, only 12 layers out of 654 are quantized. On the other hand, quantization does not seem to play a significant role in reducing inference time when *cloud* is added. Additionally, in this case the model is fully executed on the cloud, as it is the fastest hardware. Moreover, progressively adding more powerful computational resources to the system leads to lower inference times.

6.2.2 Energy Evaluation. In Fig. 3 we show the numerical values obtained from optimizing the energy consumption. Again, quantization has a beneficial effect in reducing energy consumption in both *device* and *device+edge* cases: in the first case, we move from

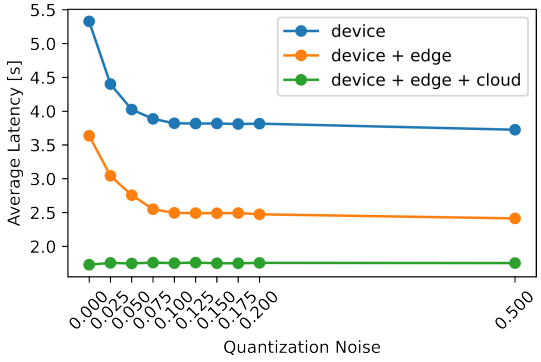


Figure 2: Latency optimization ($w_T = 1$)

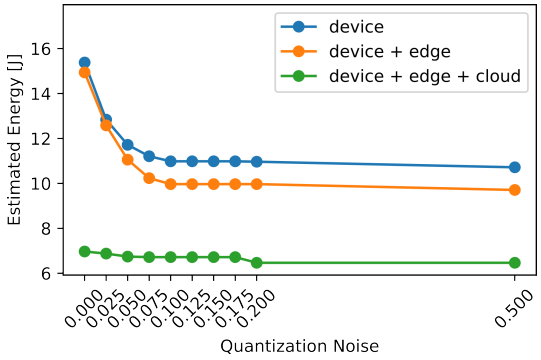


Figure 3: Energy optimization ($w_E = 1$)

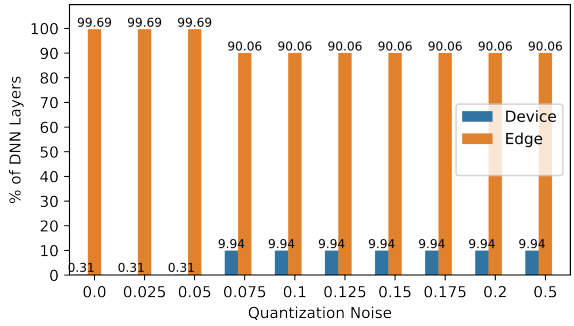


Figure 4: Model splitting: latency optimization ($w_T = 1$) in device+edge case

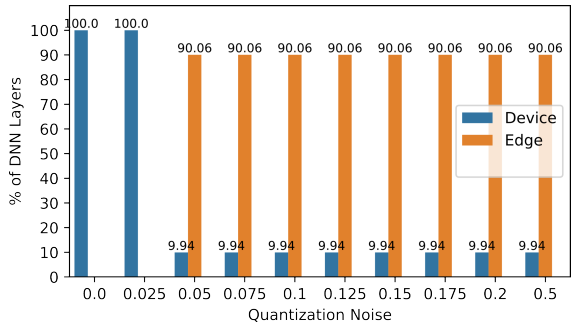


Figure 5: Model splitting: energy optimization ($w_E = 1$) in device+edge case

15.37 J to 10.71 J (a 30.3% improvement), while, in the second case, we move from 14.93 J to 9.70 J (a 35.0% improvement). In this case, when the *cloud* is added, quantization appears to slightly reduce the overall energy consumption, from 6.96 J to 6.46 J.

6.2.3 Splitting Analysis. In Fig. 4 we show an example of how the model is split in the *device+edge* case when optimizing latency. With the exception of the input and output layers, no split is applied until the maximum accepted noise reaches 0.075. Below this value, running the entire quantized model on the edge is considered more efficient; above this value, quantization reduces the size of the intermediate tensor in such a way that running the first part of the model on the device becomes more convenient.

On the other hand, in Fig. 5 we show an example of how the model is split when optimizing for energy consumption. The behavior is similar, but the system nodes used are opposite: in this case, computation is performed entirely on the device until a certain noise value is reached. The main reason for this is the high energy consumption associated with the device when transmitting: as the quantization reduces tensors sizes, transmission becomes less energy-intensive, making the splitting more convenient.

6.3 Computational Cost of ILP Resolution

The problem we presented can be seen as a graph assignment problem and, as such, it is NP-hard. We investigated the time required for ILP resolution using IBM CPLEX, generating synthetic DNN models of varying complexities (in terms of number of layers) and considering a varying number of infrastructure nodes.[1]

As we do not have a quantization noise model for synthetic models, we set the maximum quantization noise as 0. Regarding the optimization objectives, we consider the balanced optimization case with $w_T = w_E = 0.5$, which intuitively is the most complex.

Since one of the key factors affecting the problem resolution time is the DNN architecture, we built the DNNs in two ways, indicated as *static* and *random* in the following.

Static DNNs are built as sequences of identical layers (from 500 to 1,850), with the addition of input and output layers.

The resulting times for static DNNs, averaged on 5 runs, are shown in Fig. 6. As expected, the resolution time grows with both the number of servers and the number of layers in the DNN. The growth (in logarithmic scale) is close to linear. Nonetheless, we observe that the resolution times remain acceptable with a high number of layers and a moderate number of servers: for example, assuming a maximum resolution time of around 100 s, we would be able to solve the problem within this threshold for all the considered DNNs with up to 5 servers; conversely, with 6 servers we could handle would be 1252 (more than those appearing in the YOLO11 model considered in the evaluation).

Random DNNs are generated from the reference *YOLO11x-seg*. Specifically, we use *YOLO11x-seg* to extract (i) the number of FLOPs and the output size for each type of layer, and (ii) the number of

[1] In practice, "nodes" might represent computational layers or data centers.

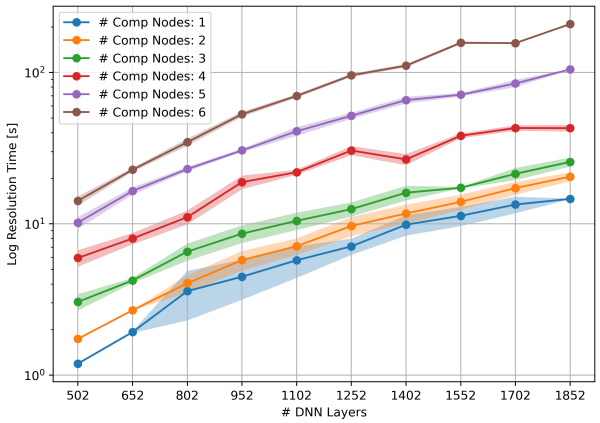


Figure 6: ILP resolution time with *static* DNNs.

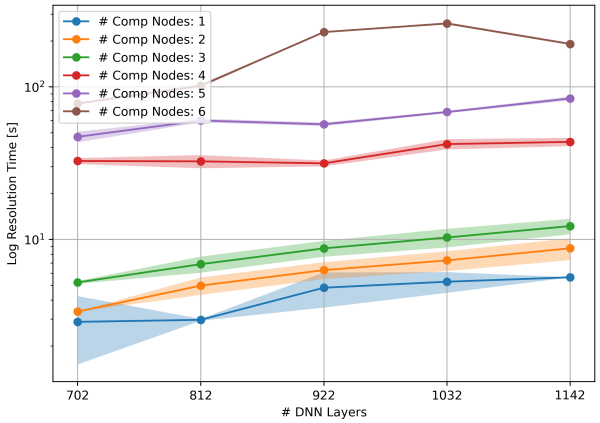


Figure 7: ILP resolution time with random DNNs.

edges connecting two layer types. Based on this, we define a uniform probability distribution for a layer of type t_1 to be connected to a layer of type t_2. We define a fixed number of parallel branches (i.e., 5) and, among these, a main branch. Then, we randomly choose the branching and merging point of each branch, and randomly add skip connections for every layer. The main branch is made up of 500 layers and the secondary of 50 layers. We increase the size of the main branch by steps of 50 layers, and the size of other branches by steps of 15 layers.

The resulting times, averaged on 5 runs and represented with a logarithmic scale, are shown in Fig. 7. While less regular than in Fig. 6, we observe a similar growth in the resolution times. The less regular trend in the times demonstrates the impact of the DNN architecture. For instance, looking at the optimization with 6 servers and with 1032 and 1142 layers, we note that the former takes more time than the latter.

7 Conclusions and Open Issues

We presented a framework and an optimization approach for the deployment of DL models across the edge–cloud continuum, with the goal of minimizing inference time and energy consumption, while modeling noise due to quantization. Experimental results on YOLO models demonstrated that the proposed framework achieves significant improvements, with reductions in inference time and energy consumption of up to 30% in certain configurations.

Motivated by these positive results, we plan to extend our approach along multiple research directions to overcome existing limitations. First, we plan to validate the proposed framework on a larger-scale testbed infrastructure, relying on real edge devices, different classes of models (e.g., LLMs), and tools to estimate the energy consumption of computing nodes (e.g., PowerAPI). At the same time, although the resolution times were acceptable in our evaluation (less than a minute in the *device+edge+cloud* configuration), we will explore heuristic strategies to overcome the scalability limitations of the exact ILP solution.

Moreover, we plan to further investigate the correlation between quantization noise and model accuracy, with the goal of translating user-specified requirements (e.g., desired accuracy) into maximum quantization noise constraints. Related to this, we also aim to extend our approach to enable quantization of more layers.

References

[1] Anaam Ansari and Tokunbo Ogunfunmi. 2019. Empirical Analysis of Fixed Point Precision Quantization of CNNs. In *Proceedings of the 2019 IEEE 62nd International Midwest Symposium on Circuits and Systems (MWSCAS '19)*. 243–246. doi:10.1109/MWSCAS.2019.8885263

[2] Amin Banitalebi-Dehkordi, Naveen Vedula, Jian Pei, Fei Xia, Lanjun Wang, and Yong Zhang. 2021. Auto-Split: A General Framework of Collaborative Edge-Cloud AI. In *Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery & Data Mining (KDD '21)*. 2543–2553. doi:10.1145/3447548.3467078

[3] Jiasi Chen and Xukan Ran. 2019. Deep Learning With Edge Computing: A Review. *Proc. IEEE* 107, 8 (2019), 1655–1674. doi:10.1109/JPROC.2019.2921977

[4] Amir Erfan Eshratifar, Mohammad Saeed Abrishami, and Massoud Pedram. 2021. JointDNN: An Efficient Training and Inference Engine for Intelligent Mobile Cloud Computing Services. *IEEE Trans. Mob. Comput.* 20, 2 (2021), 565–576. doi:10.1109/TMC.2019.2947893

[5] Yin Huang, Letian Zhang, and Jie Xu. 2026. Learning the Optimal Path and DNN Partition for Collaborative Edge Inference. *IEEE Trans. Mob. Comput.* 25, 2 (2026), 1499–1512. doi:10.1109/TMC.2025.3602966

[6] Erik Johannes Husom, Arda Goknil, Merve Astekin, Lwin Khin Shar, Andre Kåsen, Sagar Sen, Benedikt Andreas Mithassel, and Ahmet Soylu. 2025. Sustainable LLM Inference for Edge AI: Evaluating Quantized LLMs for Energy Efficiency, Output Accuracy, and Inference Latency. *ACM Trans. Internet Things* 6, 4, Article 28 (2025), 35 pages. doi:10.1145/3767742

[7] Glenn Jocher and Jing Qiu. 2024. *Ultralytics YOLO11.* https://github.com/ultralytics/ultralytics

[8] Yiping Kang, Johann Hauswald, Cao Gao, Austin Rovinski, Trevor N. Mudge, Jason Mars, and Lingjia Tang. 2017. Neurosurgeon: Collaborative Intelligence Between the Cloud and Mobile Edge. In *Proceedings of the of 22nd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '17)*. ACM, 615–629. doi:10.1145/3037697.3037698

[9] Andrey Kuzmin, Markus Nagel, Mart van Baalen, Arash Behboodi, and Tijmen Blankevoort. 2023. Pruning vs Quantization: Which is Better?. In *Proceedings of the 37th International Conference on Neural Information Processing Systems (NIPS '23, Vol. 36)*. Article 2725, 14 pages. https://proceedings.neurips.cc/paper_files/paper/2023/file/c48bc80aa5d3cbbdd712d1cc107b8319-Paper-Conference.pdf

[10] Guangli Li, Lei Liu, Xueying Wang, Xiao Dong, Peng Zhao, and Xiaobing Feng. 2018. Auto-tuning Neural Network Quantization Framework for Collaborative Inference Between the Cloud and Edge. In *Proceedings of the 27th International Conference on Artificial Neural Networks and Machine Learning, ICANN '18 (Lecture Notes in Computer Science, Vol. 11139)*. Springer, 402–411. doi:10.1007/978-3-030-01418-6_40

[11] Xiangchen Li, Saeid Ghafouri, Bo Ji, Hans Vandierendonck, Deepu John, and Dimitrios S. Nikolopoulos. 2025. QPART: Adaptive Model Quantization and Dynamic Workload Balancing for Accuracy-aware Edge Inference. *CoRR* abs/2506.23934 (2025). arXiv:2506.23934 doi:10.48550/arXiv.2506.23934

[12] Yoshitomo Matsubara, Marco Levorato, and Francesco Restuccia. 2022. Split Computing and Early Exiting for Deep Learning Applications: Survey and Research Challenges. *ACM Comput. Surv.* 55, 5, Article 90 (2022), 30 pages.

doi:10.1145/3527155
[13] Tobias Meuser, Lauri Lovén, Monowar Bhuyan, Shishir G. Patil, Schahram Dustdar, Atakan Aral, Suzan Bayhan, Christian Becker, Eyal de Lara, Aaron Yi Ding, Janick Edinger, James Gross, Nitinder Mohan, Andy D. Pimentel, Etienne Rivière, Henning Schulzrinne, Pieter Simoens, Gürkan Solmaz, and Michael Welzl. 2024. Revisiting Edge AI: Opportunities and Challenges. *IEEE Internet Comput.* 28, 4 (2024), 49–59. doi:10.1109/MIC.2024.3383758
[14] Markus Nagel, Marios Fournarakis, Rana Ali Amjad, Yelysei Bondarenko, Mart van Baalen, and Tijmen Blankevoort. 2021. A White Paper on Neural Network Quantization. *CoRR* abs/2106.08295 (2021). arXiv:2106.08295 doi:10.48550/arXiv.2106.08295
[15] Jiawei Shao and Jun Zhang. 2020. Communication-Computation Trade-off in Resource-Constrained Edge Inference. *IEEE Commun. Mag.* 58, 12 (2020), 20–26. doi:10.1109/MCOM.001.2000373
[16] Md Maruf Hossain Shuvo, Syed Kamrul Islam, Jianlin Cheng, and Bashir I. Morshed. 2023. Efficient Acceleration of Deep Learning Inference on Resource-Constrained Edge Devices: A Review. *Proc. IEEE* 111, 1 (2023), 42–91. doi:10.1109/JPROC.2022.3226481
[17] Yingchao Wang, Chen Yang, Shulin Lan, Liehuang Zhu, and Yan Zhang. 2024. End-Edge-Cloud Collaborative Computing for Deep Learning: A Comprehensive Survey. *IEEE Communic. Surv. Tutor.* 26, 4 (2024), 2647–2683. doi:10.1109/COMST.2024.3393230
[18] Yiren Zhou, Seyed-Mohsen Moosavi-Dezfooli, Ngai-Man Cheung, and Pascal Frossard. 2018. Adaptive Quantization for Deep Neural Network. In *Proceedings of the 32nd AAAI Conference on Artificial Intelligence, the 30th Innovative Applications of Artificial Intelligence, and the 8th AAAI Symposium on Educational Advances in Artificial Intelligence (AAAI'18/IAAI'18/EAAI'18)*. AAAI Press, 4596–4604. doi:10.1609/AAAI.V32I1.11623

A Post-processing Algorithm

Grouping subsequent layers by their assigned server does not guarantee an acyclic component graph. Since the optimization problem in Sec. 5 does not ensure acyclicity, we implemented an algorithm to prevent deadlocks during inference. The components of the algorithm are presented in Alg. 1.

In the `AssignNodesToComponents` function, we use topological sorting to better track dependencies, and input and output layers are placed in separate components to simplify their management. The key part of the function is the definition of the `excludeSet` for the current layer: this is the set of component identifiers that cannot be assigned to the layer, as doing so would introduce a circular dependency; specifically, a component p is excluded if there exists a component d on which layer l depends and such that a dependency exists between p and d.

In the `UpdateSets` procedure, four `for` loops are used to update different sets. The first loop updates the possible components of successor nodes, while the second updates those of parallel nodes (i.e., layers on a graph branch that is parallel to that of the current node, which are neither its descendants nor ancestors): if a successor/parallel node is assigned to the same server as the current node, the component of the latter can be a possible component for the former. This makes it possible to reduce the total number of components generated by the algorithm and to achieve better exploitation of optimizations during inference; in fact, we will have larger components, i.e., larger sub-models, on which it will be easier to apply optimizations.

The third `for` loop updates the dependencies of the layers from the components: once a node is assigned to a component, all its descendant nodes (i.e., nodes that depend directly or indirectly on its output) will depend on this component and on all components it in turn depends on.

The fourth `for` loop ensures that, if other components depend on the component to which the node has been assigned, their dependencies are updated with those of the added layer.

Algorithm 1 Assignment of Nodes to Components

Function AssignNodesToComponents(*modelGraph*, *assignmentMap*)
Initialize empty tables *nodeComponentAssignment*, *nodeDepDict*, *nodePosDict*, *compDepDict*
for each node n in *modelGraph.getNodes()* **do**
  Assign the empty set to entry n in *nodeDepDict*
  Assign the empty set to entry n in *nodePosDict*
end for
for each node n in the topological ordering of *modelGraph* **do**
  dependencySet ← set of dependencies of n from *nodeDepDict*
  possibleSet ← candidate components of n from *nodePosDict*
  Initialize *excludeSet* as an empty set
  for each component c_{dep} in *dependencySet* **do**
    for each candidate component c_{pos} in *possibleSet* **do**
      if c_{pos} is in dependencies of c_{dep} in *compDepDict* **then**
        Add c_{pos} to *excludeSet*
      end if
    end for
  end for
  differenceSet ← *possibleSet* \ *excludeSet*
  if *differenceSet* is empty **or** n is an input or output node **then**
    nodeCompId ← a new component id based on *assignmentMap[n]*
  else
    nodeCompId ← an arbitrary element of *differenceSet*
  end if
  Record in *nodeComponentAssignment* the association between n and *nodeCompId*
  UpdateSets(*modelGraph*, *assignmentMap*, n, *nodeCompId*, *nodeDepDict*, *nodePosDict*, *compDepDict*)
end for
end function AssignNodesToComponents

Procedure UpdateSets(*modelGraph*, *assignmentMap*, *n*, *nodeCompId*, *nodeDepDict*, *nodePosDict*, *compDepDict*)
if n is not an input node in *modelGraph* **then**
  for each successor node s of n in *modelGraph* **do**
    if *assignmentMap[n]* equals *assignmentMap[s]* **then**
      Add *nodeCompId* to *nodePosDict[s]*
    end if
  end for
  for each parallel node p of n in *modelGraph* **do**
    if *assignmentMap[n]* equals *assignmentMap[p]* **then**
      Add *nodeCompId* to *nodePosDict[p]*
    end if
  end for
end if
Extend *compDepDict[nodeCompId]* with *nodeDepDict[n]* \{*nodeCompId*}
for each descendant node d of n in *modelGraph* **do**
  Add *nodeCompId* to *nodeDepDict[d]*
  Extend *nodeDepDict[d]* with *compDepDict[nodeCompId]*
end for
for each component id c_{other} in the keys of *compDepDict* **do**
  if *nodeCompId* belongs to *compDepDict*[c_{other}] **then**
    Extend *compDepDict*[c_{other}] with *nodeDepDict[n]* \{*nodeCompId*}
  end if
end for
end procedure UpdateSets